

EV316937464

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Security-Related Programming Interface

Inventors:

Steven Townsend

Thomas F. Fakes

ATTORNEY DOCKET NO. MS1-1878US

1 **TECHNICAL FIELD**

2 The systems and methods described herein relate to computing systems
3 and, more particularly, to an interface associated with processing events, such as
4 security-related events, and other information.

5
6 **BACKGROUND**

7 Computer systems are continuing to grow in popularity and are frequently
8 interconnected with other computer systems via networks, such as local area
9 networks (LANs) and the Internet. Features such as electronic mail (email),
10 instant messaging, and online entertainment encourage the use of computer
11 systems coupled to networks. These features allow users to, for example,
12 communicate with other users, retrieve audio and/or video content, and purchase
13 products or services via online sources.

14 This increased interconnection of computer systems increases the
15 likelihood of attacks against the computer systems by malicious users. These
16 attacks may include installing a malicious program onto other users' computers
17 (e.g., intended to disable the other users' computers, to obtain information from
18 the other users' computers, launch attacks against other computers, and the like).
19 Attacks may also include attempting to disable a computer such that its
20 performance is greatly impaired (e.g., by generating a continuous stream of
21 requests sent to the computer). These attacks can be a nuisance to the computer
22 user and may result in lost data, corrupted data, confidential data being copied
23 from the computer, or rendering the computer inoperable.

24 To prevent or minimize the severity of such attacks, various security
25 programs and services have been developed. These programs and services execute

1 on the computer system and protect the computer system from malicious attacks.
2 Example programs include antivirus programs and firewall programs. Typically,
3 these programs or services are directed toward preventing a particular type of
4 attack. For example, an antivirus program protects against the loading and/or
5 execution of computer viruses, and a firewall program protects against
6 unauthorized access to the computer by an outside user.

7 These different programs do not typically communicate with one another.
8 For example, an antivirus program does not typically communicate the fact that a
9 virus was detected to the firewall program. Thus, the various security programs in
10 a computer system may not learn of certain attacks on the computer system. It
11 would be desirable to provide an interface that permits the communication of
12 security policies and event information among various components and security
13 programs in a computer system.

14 15 **SUMMARY**

16 The systems and methods described herein provide an interface associated
17 with processing events and other information to enhance the security of a
18 computing system. In a particular embodiment, a programming interface includes
19 a first group of functions related to communicating a new security policy to
20 multiple security engines. Each of the multiple security engines is capable of
21 replacing an existing security policy with the new security policy. The
22 programming interface also includes a second group of functions related to
23 communicating an indication of each security engine's readiness to implement the
24 new security policy.

BRIEF DESCRIPTION OF THE DRAWINGS

Similar reference numbers are used throughout the figures to reference like components and/or features.

Fig. 1 illustrates an example environment in which various events are generated and processed.

Fig. 2 illustrates an example security policy containing data and rules.

Fig. 3 illustrates an example table maintained by a security module regarding data requested by various security engines.

Fig. 4 is a flow diagram illustrating an embodiment of a procedure for retrieving and distributing security policy rules and data.

Fig. 5 is a flow diagram illustrating an embodiment of a procedure for handling updated security policy data.

Fig. 6 is a flow diagram illustrating an embodiment of a procedure for handling the distribution of information to one or more security engines.

Fig. 7 is a flow diagram illustrating an embodiment of a procedure for updating a security policy.

Fig. 8 is a flow diagram illustrating another embodiment of a procedure for updating a security policy.

Fig. 9 illustrates a general computer environment.

Figs. 10-21 illustrate various example implementations of a programming interface.

DETAILED DESCRIPTION

The systems and methods discussed herein process various information, such as events generated by one or more programs or services. Further, an interface is described that permits the communication of information, such as security-related information, among various components and programs in a computing system. The computing system includes an event manager that receives events and other information from multiple sources, such as security engines and other computing systems. Example security engines include antivirus engines, firewall engines and intrusion detection engines. The event manager communicates event information received from a particular source to one or more security engines that might use the information to improve the level of security provided for the computing system.

Although particular examples discussed herein refer to security-related events and other security-related information, alternate embodiments may process any type of event or information. This information includes any information that might be utilized by security-related components in a host computer. Alternate embodiments can receive, process and distribute information that is not necessarily related to the security of the host computer. The terms “interface”, “program interface” and “application program interface (API)” are used interchangeably herein.

Event Processing

Fig. 1 illustrates an example environment 100 in which various events are generated and processed. Events include, for example, detection of a computer virus, detection of an attempt to access confidential data, notification that a

1 computer virus was destroyed, notification that a particular application program
2 was halted or prevented from executing, changes to system state information, and
3 so forth. A host computer 102 is coupled to multiple servers 104 and 106 via a
4 network 108. Host computer 102 and servers 104 and 106 may be any type of
5 computing device, such as the device discussed below with respect to Fig. 9.

6 Network 108 can be any type of data communication network, such as a local area
7 network (LAN), wide area network (WAN), the Internet, and the like. Although
8 Fig. 1 show host computer 102 coupled to two servers 104 and 106, host computer
9 102 may be coupled to any number of servers or other devices capable of
10 communicating with the host computer.

11 Environment 100 can represent any of a variety of a settings, such as
12 networks in home, business, educational, research, etc. settings. For example,
13 server 104 may be a server device on a corporate LAN, and host computer 102
14 may be a desktop or portable computing device on the corporate LAN. By way of
15 another example, server 104 may be a server device on the Internet, and host
16 computer 102 may be a desktop computing device at a user's home.

17 Host computer 102 includes a security module 110 that performs various
18 security-related functions, such as monitoring, detecting and responding to attacks
19 on host computer 102. Security module 110 includes an event manager 112 that is
20 coupled to three security engines 114, 116 and 118. A security engine can be any
21 service that assists in protecting against malicious users and/or malicious
22 programs. Security engines 114-118 may be implemented in software, hardware,
23 or a combination of software and hardware. Particular security engines are
24 security-related application programs, such as antivirus programs and intrusion
25 detection programs. Security engines 114-118 may also be referred to as

1 “services”. A particular security module 110 may include any number of security
2 engines coupled to event manager 112. Security module 110 may also include
3 other modules, components, or application programs (not shown), such as a
4 security-related policy reader or other policy-handling mechanism.

5 Security module 110 is also coupled to system state information 120 and
6 system configuration information 122. System state information 120 includes
7 information regarding the current operating state or operating mode of host
8 computer 102. System configuration information 122 includes information
9 regarding how host computer 102 is configured. System state information 120 and
10 system configuration information 122 may be stored in a non-volatile storage
11 device, such as a memory device or a hard disk drive. In one embodiment, event
12 manager 112 and security engines 114-118 are capable of receiving system state
13 information 120 and system configuration information 122.

14 Host computer 102 also includes an application program interface (API)
15 124 that permits the communication of security policies and event information
16 among various components and programs in host computer 102 or other devices.
17 For example, API 124 allows components or programs to communicate with
18 security engines 114-118 or event manager 112 to send or receive security-related
19 information. API 124 also facilitates, for example, loading new security engines,
20 unloading existing security engines, sending security policies to security engines,
21 communicating changes in data to security engines, user interaction with security
22 engines, and centralized configuration management of security engines.
23 Additional details regarding API 124 are discussed below.

24 Although not shown in Fig. 1, additional data sources or data providers may
25 communicate information and events to security module 110 and event manager

1 112. This additional data includes, for example, configuration information related
2 to an Internet Information Service (IIS), data provided by an system management
3 application, data contained in a system registry, and information provided by a
4 user or administrator of the system.

5 Each security engine 114-118 performs certain security-related functions to
6 help secure host computer 102 from malicious users or application programs.

7 These malicious users or application programs may attempt to disable host
8 computer 102 or disable functionality of host computer 102, obtain data from host
9 computer 102 (such as passwords or other confidential information), or use host
10 computer 102 (such as to assist in attacking other computer systems). For
11 example, security engine 114 detects computer viruses, security engine 116
12 provides firewall protection, and security engine 118 blocks execution of
13 particular application programs based on one or more user privileges or
14 characteristics. In this example, security engine 114 protects host computer 102
15 from being infected by computer viruses, worms, Trojan horses, and the like.

16 Additionally, firewall protection includes protecting host computer 102 from being
17 accessed over a network connection by other devices. Blocking execution of
18 particular application programs includes preventing execution of application
19 programs on host computer 102 by a user that does not have appropriate
20 privileges. Additionally, execution of an application program may be blocked if
21 improper behavior is detected, such as improper network access or improper
22 storage device access.

23 In other embodiments, one or more security engines may perform intrusion
24 detection or vulnerability analysis. Intrusion detection includes, for example,
25 identifying when a malicious application program and/or user has accessed host

1 computer 102 and taking appropriate action to notify a user or administrator,
2 attempt to disable the malicious application program, or halt the malicious user's
3 access. Vulnerability analysis includes, for example, attempting to detect
4 vulnerabilities in host computer 102 due to security engines or other components
5 that have not been installed or updated correctly, security engines or other
6 components that have not been configured properly, patches or hot fixes that have
7 not been installed, passwords that do not comply with required lengths or required
8 characters, and the like. A particular security engine 114-118 may be unaware of
9 the existence and functionality of other security engines coupled to event manager
10 112.

11 Each security engine 114-118 communicates events (e.g., detection of a
12 computer virus, detection of an attempt to retrieve data from host computer 102, or
13 preventing execution of an application program by a user) to event manager 112.
14 These events include information collected by a security engine, actions taken by a
15 security engine, data collected by the event manager from one or more data
16 sources, and the like. Example information includes a listing of all virtual servers
17 instantiated in a particular installation. Event manager 112 processes these events
18 and communicates the information contained in particular events to other security
19 engines 114-118 that may benefit from such information.

20 Security module 110 also receives security-related policies that include one
21 or more rules and various data. Event manager 112 distributes the rules to the
22 appropriate security engines 114-118 and provides data to the security engines, as
23 needed. Each security engine 114-118 stores these rules and data received from
24 event manager 112. The operation of security module 110, event manager 112 and
25 security engines 114-118 is discussed in greater detail below.

1 Fig. 2 illustrates an example security policy 200 containing data and rules.
2 In one embodiment, security policy 200 is stored in security module 110. A
3 particular security module may receive and store any number of different security
4 policies 200 received from any number of different data sources. Alternatively,
5 security policy 200 may be stored in another module or component within host
6 computer 102. In the example of Fig. 2, a data portion 202 of security policy 200
7 includes one or more data elements. As shown in Fig. 2, these data elements
8 include values assigned to variables (e.g., a value of “1” is assigned to variable
9 “A” and a value of “4” is assigned to variable “B”). In alternate embodiments,
10 other types of data may be included instead of or in addition to the data shown in
11 Fig. 2. The data contained in security policy 200 is used, for example, by one or
12 more rules contained in security policy 200 or contained in one or more other
13 security policies.

14 Security policy 200 also includes a rules portion 204 that contains multiple
15 rules. The rules in security policy 200 may be associated with one or more
16 security engines. For example, certain rules may only be applied by particular
17 security engines. The rules may be arranged in security policy 200 based on the
18 security engine with which the rules are associated. Alternatively, an identifier
19 associated with each rule may identify the security engines that are capable of
20 applying the rule. In particular embodiments, a rule may be associated with any
21 number of security engines. In other embodiments, a host computer may not
22 contain a security engine that applies a particular rule. In this situation, the rule is
23 not associated with any security engine.

24 In the example of Fig. 2, the rules are defined using an IF-THEN structure.
25 Alternatively, the set of rules can take a variety of different forms. Using the IF-

1 THEN structure shown in Fig. 2, the rule defines a particular condition(s) and a
2 corresponding particular action(s) or result(s). During enforcement of the rule, if
3 that particular condition(s) is detected, then the corresponding particular action(s)
4 or result(s) is performed. A rule can identify a variety of different conditions and
5 corresponding actions or results. Example conditions include attempts to access a
6 resource (e.g., memory locations, network addresses or ports, other programs, or
7 files on a storage device), attempts to write data to particular locations (e.g.,
8 particular memory locations, or particular locations on a storage device), attempts
9 to run particular programs, and various aspects of the current operating state of
10 host computer 102. Example results include preventing a resource from being
11 accessed, preventing data from being written to particular locations, preventing a
12 program from being executed, or generating a notification that the occurrence of
13 the condition in the rule was detected (e.g., recording its occurrence in a log, or
14 sending a message to a user or other computer). The particular results can also be
15 permissive in nature rather than preventive. For example, the results could
16 indicate that a particular resource or location can be accessed only if the condition
17 in the rule is satisfied by host computer 102, or that a particular program can only
18 be run if the condition in the rule is satisfied by host computer 102.

19 Additional examples of rules include permitting certain application
20 programs or services to update data files in a particular directory or folder,
21 enabling receipt of traffic on port 21 if file transfer protocol (FTP) is enabled, and
22 generating a virus warning message if a particular virus signature is detected.
23 Other examples include generating an event if a particular application program has
24 not been upgraded to a particular revision level, preventing access to a network if
25

1 the application program has not been upgraded to a minimum revision level, and
2 preventing the host computer from receiving data via network port 35.

3 Fig. 3 illustrates an example table 300 maintained by a security module
4 regarding data requested by various security engines. In one embodiment, table
5 300 is stored in security module 110. Alternatively, table 300 may be stored in
6 another module or component within host computer 102. Each time a security
7 engine requests data from the security module, the security module updates the
8 table (if necessary) to include that data request. A first column 302 of table 300
9 identifies a particular data element, such as a variable or other identifier or
10 information. A second column 304 of table 300 identifies any security engines
11 that previously requested the associated data element. For example, table 300
12 identifies that data element "A" was previously requested by security engine "1".
13 Similarly, data element "D" was previously requested by security engines "1", "4"
14 and "6". As discussed in greater detail below, the information contained in table
15 300 is used by the security module to determine which security engines should
16 receive updated data.

17 Fig. 4 is a flow diagram illustrating an embodiment of a procedure 400 for
18 retrieving and distributing security policy rules and data. Procedure 400 may be
19 performed, for example, upon initialization of a host computer. Initially, a security
20 module retrieves security policies for the host computer (block 402). A event
21 manager identifies rules in the security policies related to each security engine
22 (block 404). The event manager then communicates the rules to the appropriate
23 security engines (block 406).

24 Each security engine identifies data necessary to apply its associated rules
25 (block 408), for example by identifying data elements contained in rules that the

1 security engine will apply. Each security engine then requests its identified data
2 from the event manager (block 410). After receiving a data request from a
3 security engine, the event manager records the requested data element in a table
4 (e.g., table 300 in Fig. 3) or other data structure for future reference (block 412).
5 Finally, the event manager locates the requested data and provides that data to the
6 requesting security engine (block 414). Thus, rather than providing all data to all
7 security engines, the event manager provides the requested data to each requesting
8 security engine.

9 Fig. 5 is a flow diagram illustrating an embodiment of a procedure 500 for
10 handling updated security policy data. Initially, the security module receives
11 updated data (block 502). For example, the updated data may include updated
12 values for existing variables. The security module identifies one or more security
13 engines that previously requested the data that has been updated (block 504). In
14 one embodiment, the security module identifies these security engines using a
15 table such as table 300 shown in Fig. 3. After identifying the appropriate security
16 engines, the security module provides the updated data to each of the identified
17 security engines (block 506). Finally, the security engines update their data
18 elements with the updated data. Procedure 500 is repeated each time the security
19 module receives updated data. In another embodiment, the security module
20 periodically checks various data sources for updated data. If the data has been
21 updated, the security module retrieves the updated data and distributes the data
22 according to procedure 500.

23 In one embodiment, when a rule is updated, the security module identifies
24 the security engines associated with the rule and distributes the updated rule to the
25 identified security engines. If a new rule is received, the security module

1 identifies the security engines that might use the new rule and distributes the new
2 rule to the appropriate security engines. Similarly, if an existing rule is deleted,
3 the security module deletes the rule from all security engines associated with the
4 rule. In another embodiment, when a rule is updated, the security module creates
5 a new set of rules (including the updated rule) and distributes the new set of rules
6 to the security engines, thereby replacing the existing rules contained in the
7 security engines.

8 Fig. 6 is a flow diagram illustrating an embodiment of a procedure 600 for
9 handling the distribution of information, such as event information or system state
10 information, to one or more security engines. Initially, the event manager receives
11 an event from a security engine (block 602). The event manager then identifies
12 information contained in the event (block 604), such as the event type or the
13 nature of the attack that generated the event. The event manager also identifies
14 other security engines that might use the information contained in the event (block
15 606). The relationships among different security engines are specified, for
16 example, in the security policy received by the host computer. These relationships
17 may be defined wholly or in part by a system administrator or other system
18 operator when creating the security policy.

19 Next, the event manager provides the information contained in the event to
20 the identified security engines (block 608). The identified security engines then
21 apply the received information (block 610). This sharing (or correlation) of event
22 information enhances the level of security provided by a host computer against
23 malicious attacks. Sharing of the event information is handled by the event
24 manager such that the individual security engines do not need to know of the other
25 security engines contained in the host computer. The security-related information

1 discussed herein can be stored in a central location, thereby allowing other
2 devices, components and application programs to access the information. For
3 example, other security engines or computing systems may access the stored
4 security related information.

5 In one example of procedure 600, an antivirus security engine detects
6 repeated attempts to access a network via a particular port. The antivirus security
7 engine reports this information (e.g., dates and times of the attempted access and
8 the port on which access was attempted) to the event manager. In this example,
9 the antivirus security engine is not responsible for responding to such access
10 attempts. The event manager receives the information from the antivirus security
11 engine and determines that an intrusion detection security engine and a firewall
12 security engine may use such information. After receiving the information, the
13 intrusion detection security engine and the firewall security engine may adjust
14 their operation based on the received information. For example, the intrusion
15 detection security engine may increase the frequency with which it checks for
16 intruders. Additionally, the firewall security engine may temporarily disable the
17 port on which access was attempted. Thus, the overall security of the host
18 computer against attacks is increased by allowing security engines to adjust their
19 operation based on shared information regarding security-related events.

20 In another example of procedure 600, a vulnerability security engine detects
21 whether a particular patch is installed on the host computer. If the patch is not
22 installed, the vulnerability security engine generates an event indicating that the
23 patch is not installed. A host firewall security engine and a behavioral blocking
24 security engine have registered with the event manager for notification if the patch
25 is not installed. When the host firewall security engine and the behavioral

1 blocking security engine receive notification of the patch not being installed, the
2 security engines enforce rules that limit the functionality (or prevent execution) of
3 the application program that was not patched.

4 In another embodiment, system state information is shared among various
5 components (e.g., the event manager and multiple security engines) in the security
6 module. The system state information may be provided by various data sources.
7 Example system state information includes a current network state, whether a
8 network connection is wired or wireless, whether the host computer is accessing a
9 corporate network or an unknown network, and host computer configuration
10 information. Thus, if a security engine identifies particular system state
11 information, that identified information can be shared among other security
12 engines and other components or modules in the host computer.

13 In a particular embodiment, the system state information collected by
14 various components is stored in a central location, thereby providing access to the
15 information by other devices, components and application programs. For
16 example, system state information collected by one security engine is accessible
17 by other security engines, security modules and computing systems.

18 19 Security Policy Updates

20 As discussed above, a security policy can be used to describe the rules that
21 are to be applied, for example, by security engines or security providers. When
22 changes are made to the security policy, updated rules are supplied to the various
23 security engines, and these various security engines change over to start using the
24 updated rules at substantially the same time.

1 In addition to the components, programs and modules discussed above with
2 respect to Fig. 1, host computer 102 receives security policies from one or more
3 source devices, such as servers 104 and 106. The security policies describe how
4 various security engines on host computer 102 should operate. Although only one
5 host computer 102 is illustrated in Fig. 1, it is to be appreciated that multiple host
6 computers 102 can obtain security policies from the same source device.

7 Example source devices include desktop or workstation computing devices,
8 server computing devices, portable or handheld computing devices, game
9 consoles, network appliances, cellular phones, personal digital assistants (PDAs),
10 networking devices (e.g., routers, gateways, firewalls, wireless access points, etc.),
11 and so forth.

12 Host computer 102 may also include a policy reader module, a rule
13 manager, a rule set generator module, and a dynamic rules data store. It is to be
14 appreciated that one or more of these modules may be combined into a single
15 module, and/or one or more of these modules may be separated into two or more
16 modules.

17 Generally, to update the security policy being enforced by security engines
18 114-118, the policy reader obtains a security policy from a source device. The
19 rule set generator uses the newly obtained security policy to generate, for each of
20 the various security engines, a set of one or more rules and associated data. These
21 sets of rules are then communicated to the various security engines, and the
22 associated data is stored in the dynamic rules data store. The associated data can
23 also be communicated to the security engines. Upon receiving the set of one or
24 more rules, each security engine processes the new set of rules, getting ready to
25 begin using the new set of rules. However, each security engine continues to use

1 its current set of rules until instructed to change to the new set. A rule manager
2 instructs all of the security engines to change to the new set of rules after the rule
3 manager receives an indication from each of the security engines that it is ready to
4 change to the new set of rules.

5 In certain embodiments, the rule manager coordinates the updating of
6 security policies in host computer 102. The rule manager receives the indications
7 from the various security engines that indicate the security engines are ready to
8 change to the new set of rules, and gives an indication to the security engines
9 when they should begin using the new set of rules.

10 The policy reader module obtains a new security policy from the source
11 device. The policy reader module may be configured to check whether a new
12 security policy is available from the source at regular or irregular intervals, or
13 alternatively may receive an indication from some other component (e.g., the rule
14 manager, the source device, or some other device not shown in Fig. 1, that it
15 should obtain a new security policy from the source (or check whether a new
16 security policy is available from the source). The policy reader may identify to the
17 source a particular security policy that the policy reader desires to obtain, or
18 alternatively may merely request the most recent security policy for the host
19 computer from the source. A comparison between the current security policy
20 being used by the host computer and the most recent security policy may be made
21 to determine whether the most recent security policy is already being enforced on
22 the host computer. Such a comparison could be made by the source, the policy
23 reader, or alternatively by some other component.

24 When the new security policy is obtained from the source, the rule set
25 generator generates a set of rules for each of the different security engines 114-

1 118. Different security engines may use different rules when enforcing the
2 security policy on host computer 102. For example, one security engine 114 may
3 be a firewall whereas another security engine 116 may be an antivirus component.
4 The security policy may identify rules that are specific to the antivirus engine (and
5 thus the firewall engine need not be concerned with them), and may also identify
6 rules that are specific to the firewall engine (and thus the antivirus engine need not
7 be concerned with them).

8 In certain embodiments, the security policy itself is a list of rules and
9 associated data. The security policy may also include an indication of which rules
10 and data are for which security engines, or alternatively no such indication may be
11 included (e.g., relying on the host computer to determine which rules are for which
12 security engines). The security policy allows designers to have a single record or
13 file of all the rules involved in the protection of the host computer, without having
14 to separate the rules across different records or files for the different security
15 engines.

16 Additionally, using the techniques described herein, new security policies
17 can be prepared by designers that shift responsibility for protecting against
18 particular attacks from one security engine to another. For example, protection
19 against a particular type of attack may be enforced by an antivirus program in one
20 security policy but changed to being enforced by a firewall program in a new
21 security policy. Using the techniques described herein, the designers can be
22 confident that this shift in responsibility will occur in all of the security engines
23 substantially concurrently, thereby reducing the vulnerability of the host computer
24 to attacks during the shift.
25

1 The rule set generator identifies, based on the security policy, which rules
2 and associated data (if any) are used by which of the security engines. For each
3 security engine, the rule set generator generates a set of rules for that security
4 engine and makes that generated set of rules available to that security engine (e.g.,
5 the set of rules may be transmitted or sent to the security engine, the security
6 engine may be informed of a location in memory where the generated set of rules
7 can be obtained, etc.). This generation can be performed in a variety of different
8 manners. For example, a new set of rules may be generated by the rule set
9 generator without regard for the current rules being enforced by the security
10 engines. By way of another example, the current set of rules may be modified or
11 changed to incorporate any differences between the current and new set of rules.
12 Additionally, the rule set generator may simply copy the rules from the security
13 policy, or alternatively the rule set generator may generate the rules based on
14 information in the security policy that describes the rules.

15 In certain embodiments, the security policy identifies which rules are to be
16 distributed to which security engines. For example, each rule may be associated
17 with a particular label or identifier (e.g., Security Engine 1, or Antivirus engine,
18 etc.). The rule set generator can use these identifiers in generating the sets of rules
19 for the various security engines. In alternate embodiments, the rule set generator
20 may infer which rules are to be distributed to which security engines. In other
21 embodiments, a combination of these techniques may be used (e.g., for some rules
22 the security policy may identify which security engine they are to be assigned to,
23 and for other rules the security policy generator may infer which security engine
24 they are to be assigned to).

1 The set of rules generated by the rule set generator can take any of a variety
2 of different forms. In certain embodiments, the rules follow an if-then structure as
3 discussed above. Using this structure, the rule defines a particular condition(s)
4 and a corresponding particular action(s) or result(s). During enforcement of the
5 rule, if that particular condition(s) is detected then the corresponding particular
6 action(s) or result(s) is performed. Any of a variety of conditions and
7 corresponding results can be identified by a rule. Examples of particular
8 conditions include: attempts to access particular resources (e.g., memory
9 locations, network addresses or ports, other programs, files on a storage device,
10 and so forth), attempts to write data to particular locations (e.g., to particular
11 memory locations, to particular locations on a storage device, etc.), attempts to run
12 particular programs, various aspects of the current operating state of the host
13 computer (e.g., resources available, programs running, etc.), and so forth.
14 Examples of particular results include: preventing a resource from being accessed,
15 preventing data from being written to particular locations, preventing a program
16 from being run, generating a notification that the occurrence of the condition in the
17 rule was detected (e.g., recording its occurrence in a log, sending a message to a
18 user or other computer, and so forth). The particular results can also be permissive
19 in nature rather than preventive. For example, the results could indicate that a
20 particular resource or location can be accessed only if the condition in the rule is
21 satisfied by the host computer, or that a particular program can be run only if the
22 condition in the rule is satisfied by the host computer.

23 In certain embodiments, host computer 102 includes a dynamic rules data
24 store which is the data associated with the various rules being enforced by the
25 security engines. In certain embodiments, the dynamic rules data store may

1 include two sets of data: one set for the current rules being enforced by the
2 security engines, and another set for the new rules that the security engines are
3 being updated to enforce. When a new security policy is received, the rule set
4 generator updates the dynamic rules data store with the data associated with the
5 sets of new rules passed to the security engines.

6 Each security engine includes a rule change module that receives a set of
7 one or more rules from the rule set generator. The data associated with the rules
8 may be received from the rule set generator along with the rules, or alternatively
9 the rule change module may obtain the data it desires from the dynamic rules data.
10 Additionally, it should be noted that although the rule set generator is discussed
11 above as generating a set of rules for each security engine based on the security
12 policy, alternatively each security engine may receive the entire security policy (or
13 most of the security policy) and generate their own set of rules rather than
14 receiving the set from the rule set generator.

15 The rule change module processes the new set of rules as needed in order to
16 generate new internal rules that enforce the new policy. The processing of the
17 new set of rules to generate new internal rules refers to whatever actions are
18 necessary for the security engine to take in order to place the new set of rules in a
19 state that they can be enforced by the security device. For example, this
20 processing may include converting the new set of rules to an internal format,
21 storing rules in particular memory locations, organizing rules into a particular
22 arrangement or order, etc. The rule change module may generate new rules in any
23 of a variety of manners; the rule change module may keep the rules in the same
24 format as they were received from the rule set generator or alternatively convert
25 the rules to an internal format use by the security engine.

1 Regardless of how the new rules are generated, each security engine
2 maintains a current set of rules which enforce the previous security policy for the
3 host computer (the security policy which is being updated). While generating the
4 new rules, and even after the new rules are generated, the security engine
5 continues to enforce the current rules. The security engine does not begin
6 enforcing the new rules until instructed to do so (e.g., by the rule manager).

7 After the rule change module has finished generating the new rules, the
8 rule change module indicates to the rule manager that it has finished and is ready
9 to switch to using the new rules (and thus begin enforcing the new security
10 policy). After the rule manager has received such an indication from all of the
11 security engines, the rule manager instructs each of the security engines to begin
12 using the new rules. The rule manager waits to instruct each of the security
13 engines to begin using the new rules until after the rule manager receives the
14 indication from all of the security engines. Once instructed to do so, each security
15 engine begins using the new rules. As soon as a security engine begins using the
16 new rules, it can delete the rules it was previously using.

17 In some situations, a security engine may fail in processing the new rules.
18 In such situations, the security engine returns an indication of such failure to the
19 rule manager. Alternatively, the rule manager may impose a time limit on
20 responses from the security engines. If all security engines have not responded
21 with an indication that they are ready to begin using the new rules within the time
22 limit, the rule manager can assume that one or more of the security engines has
23 failed in processing the new rules.

24 When the rule manager identifies that one or more of the security engines
25 has failed in processing the new rules, the rule manager does not instruct any of

1 the security engines to begin using the new rules. Rather, the rule manager sends
2 an indication to abort the changeover to the new rules (this may also be referred to
3 as a rollback). Such an abort or rollback indication informs each of the security
4 engines that it is to ignore the new rules received from the rule set generator as
5 well as any new rules resulting from its processing, and continue to use the current
6 rules. In certain embodiments, the security engines can safely delete the new rules
7 they generated (or are in the process of generating) in response to such an abort or
8 rollback indication.

9 In certain embodiments, each security engine waits until it can nearly
10 ensure that it can begin using the new rules before informing the rule manager that
11 it is ready to begin using the new rules. In other words, the security engine waits
12 to inform the rule manager that it is ready to begin using the new rules until the
13 security engine is to the point in processing the new rules that it is virtually
14 impossible for the security engine to fail to begin enforcing those rules when
15 instructed to do so. In certain embodiments, this is accomplished by the security
16 engine generating the new set of rules, and maintaining a pointer to which of the
17 rule sets (old rules or new rules) it is to use. After the new set of rules is
18 generated, the security engine indicates to the rule manager that the security
19 engine is ready to begin using the new set of rules. Then, when instructed to begin
20 using the new set of rules, the security engine can simply change its pointer from
21 the old set of rules to the new set of rules. The security engine can nearly ensure
22 that it can change its pointer and begin using the new rules. It is to be appreciated
23 that "nearly ensure" does not require a 100% guarantee that failure is absolutely
24 impossible. It is possible that certain situations could still arise that would result
25

1 in failure (e.g., a power loss or virus attack that prohibits changing of the pointer).

2 However, it is also to be appreciated that the chances of failure are very small.

3 The rule manager can instruct the security engines to begin using the new
4 set of rules (also referred to as switching over to the new set of rules) in any of a
5 variety of different manners. The manner that is used, however, should operate to
6 inform all of the security engines at substantially the same time so that all of the
7 security engines can begin using their new sets of rules at substantially the same
8 time (also referred to herein as substantially concurrently). By having all of the
9 security engines begin using their new sets of rules at substantially the same time,
10 any vulnerability of the host computer due to the rule changeover is reduced.

11 Generally, the closer in time that the security engines begin using their new sets of
12 rules, the lesser the vulnerability during the changeover to the new set of rules.

13 Following are some examples of ways in which the rule manager can instruct the
14 security engines at substantially the same time to begin using their new sets of
15 rules.

16 One way in which the rule manager can instruct the security engines to
17 begin using the new set of rules is to use an event object that can be fired across all
18 of the security engines at once. For example, each security engine, upon receipt of
19 the new rules from the rule set generator, sets an internal flag to start polling a
20 particular event each time the rules are accessed (during its normal operation of
21 protecting the host computer). The rule manager can then instruct the security
22 engines to begin using their new sets of rules by firing the event (the same one
23 being polled by the security engines). So, after the event is fired, any subsequent
24 polling of the event will reflect that the event has been fired and thereby inform
25 the polling security engine that the new rule set should be used. For example, in

1 response to detecting the event having been fired, the pointer in the security engine
2 can be changed to point to the new set of rules.

3 In addition to polling the event, a thread may also be run by the security
4 engine that waits on the event. When the event is fired, the thread detects the
5 firing so that the security engine is informed that the new rule set should be used.
6 For example, in response to the thread detecting that the event has fired, the
7 pointer in the security engine can be changed to point to the new set of rules.

8 Once the event has been fired and the new set of rules is being used, the
9 security engine can stop polling the event. Additionally, if a thread waiting on the
10 event is run by the security engine, that thread can be terminated.

11 Another way in which the rule manager can instruct the security engines to
12 begin using the new set of rules is to call a function exposed by each of the
13 security engines (e.g., a "switch" function). Calling such a function of a security
14 engine instructs that security engine to begin using the new set of rules. For
15 example, in response to such a function being invoked on a security engine, the
16 security engine changes its pointer to point to the new set of rules.

17 Another way in which the rule manager can instruct the security engines to
18 begin using the new set of rules is to notify each of the security engines of a
19 shared data structure that each security engine can access. The rule manager can
20 inform each security engine of the shared data structure at different times, such as
21 by calling a function on each security engine (e.g., an "identify data structure"
22 function), or by identifying the shared data structure when the new rules are
23 passed to the security engine. The shared data structure can take any of a variety
24 of different forms, such as a location in memory (e.g., in random access memory
25

1 (RAM) or a nonvolatile memory such as Flash memory), a file stored on a local or
2 remote storage device, and so forth.

3 Each security engine checks this shared data structure (e.g., each time the
4 rules are accessed (during its normal operation of protecting the host computer)) to
5 determine its value. The rule manager can instruct each of the security engines to
6 begin using the new rule set by changing the value(s) stored in the shared data
7 structure. For example, the shared data structure may initially store a value of
8 "previous" or a value of 0 to indicate that the current set of rules are to be used,
9 and when it is time to switch to begin using the new rule set the rule manager can
10 write a value of "new" or "switch" or a value of 1 to the shared data structure to
11 indicate that the new set of rules are to be used.

12 As discussed above, the dynamic rules data store stores the data associated
13 with the various rules being enforced by the security engines. As such, when the
14 host computer is being updated to begin enforcing a new policy, the data used by
15 the security engine may also change. This data can also change during the
16 operation of the host computer (e.g., a security engine may later request data from
17 or store data in the dynamic rules data store). In order for the proper data to be
18 made available to the security engines, when updating the security policy the
19 dynamic rules data store may operate in the same manner as a security engine.
20 That is, two sets of rules data would be maintained – the first set would be used
21 prior to the switch and the second set would be used after the switch. The new
22 data would be stored in the dynamic rules data store, and the dynamic rules data
23 store would return an indication to the rule manager when it is ready to begin
24 using the new set of data. The dynamic rules data store then continues to use the
25

1 previous set of data until receiving an instruction from the rule manager to begin
2 using the new set of data.

3 It should be noted that the various components in the host computer can be
4 implemented within the same application process executing on the host computer.
5 For example, the policy reader, the rule manager, the dynamic rules data, the rule
6 set generator, and the security engines may all be part of the same application
7 process.

8 Alternatively, different components in the host computer can be
9 implemented across two or more application processes executing on the host
10 computer. For example, one or more security engines may run in a process that is
11 separate from the other security engines as well as separate from the policy reader,
12 the rule manager, the dynamic rules data, and the rule set generator. Allowing
13 different components to run in different application processes allows, for example,
14 different developers to design different plug-in components (e.g., different plug-in
15 security engines) to enhance the security of the host computer. These additional
16 plug-in components would be upgraded to enforce new policies in the same
17 manner as other non-plug-in components.

18 When separating the components across multiple application processes, a
19 mechanism is used to allow the various components to communicate with one
20 another. This communication allows, for example, sets of new rules and data to be
21 passed to security engines in different processes, data to be passed from security
22 engines in different processes to the dynamic rules data, instructions to begin
23 using the new sets of rules to be passed to security engines in different processes,
24 and so forth. By way of example, the components discussed herein may be
25 implemented as Component Object Model (COM) components. Additional

1 information regarding the Component Object Model architecture is available from
2 Microsoft Corporation of Redmond, Washington.

3 It should be noted that in the discussions herein, each security engine is
4 instructed to begin using its new set of rules by the rule manager. Alternatively,
5 this instruction may be implemented in other manners that still allow each security
6 engine to begin using its new set of rules substantially concurrently. For example,
7 rather than using the rule manager, a control mechanism to instruct each security
8 engine to begin using its new set of rules may be distributed across the various
9 security engines. This could be accomplished, for example, by each of the
10 security engines notifying each other security engine that it is ready to begin using
11 the new set of rules, with none of the security engines beginning to use its new set
12 of rules until all of the security engines have notified all of the other security
13 engines that they are ready to begin using the new set of rules.

14 Fig. 7 is a flowchart illustrating an example process 700 for updating a
15 security policy. Process 700 is implemented by a component(s) that is
16 coordinating the updating of the security policy on a host computer, such as the
17 rule manager discussed herein. Process 700 may be performed in software,
18 hardware, firmware, or combinations thereof.

19 Initially, a new policy to be enforced on the device is obtained (block 702).
20 The policy may be obtained in a "pull" manner, where the host computer initiates
21 the access to the source of the new policy to check whether a new policy is
22 available from the source. The policy may alternatively be obtained in a "push"
23 manner, where the host computer is informed of (e.g., sent a message or other
24 indication of) the availability of a new security policy or of the new security policy
25 itself.

1 Regardless of how the new policy is obtained, once the new policy is
2 obtained a new set of rules and/or data associated with the rules for the new policy
3 is provided to each of the security devices (block 704). As discussed above,
4 different sets of rules and/or data can be generated based on the new policy for
5 each security engine.

6 Return values are then received from the security engines (block 706). In
7 certain implementations, each security engine returns, to the component
8 implementing process 700, a value signifying "OK" or a value signifying "Fail".
9 When a security engine returns a value signifying "OK" it indicates that the
10 security engine has processed the set of rules and/or data that it received and is
11 ready to begin using the new set of rules and/or data. This may also be referred to
12 as a security engine's readiness to implement the new set of rules and/or data. For
13 example, all that remains is for the security engine to change its pointer to point to
14 the new set of rules rather than the previous set of rules. However, when a
15 security engine returns a value signifying "Fail", it indicates that the security
16 engine could not (or did not) process the set of rules and/or data and that the
17 security engine is not able to begin using the new set of rules and/or data.
18 Additionally, as discussed above a time limit (also referred to as a timeout value or
19 a threshold amount of time) may be imposed on responses from security engines –
20 if a security engine does not respond with a value signifying "OK" or "Fail" within
21 this time limit the component implementing process 700 treats the security engine
22 as if it had returned a value signifying "Fail".

23 It is to be appreciated that the sending of rules and the receiving of
24 responses (blocks 740 and 706) is an asynchronous process. Different security
25 engines may take different amounts of time to process the rules and/or data they

1 receive, and process 700 simply waits until all of the security engines have
2 finished their respective processing (up to any optional time limit that is imposed).

3 Process 700 proceeds based on whether all of the security engines have
4 returned a value signifying "OK" (block 708). If all of the security engines have
5 returned a value signifying "OK", then all of the security engines are ready to
6 begin using the new set of rules, so all of the security engines are instructed to
7 begin using the new set of rules block 710).

8 However, if at least one of the security engines does not return a value
9 signifying "OK", then a rollback call is issued to each security engine (block 712).
10 This rollback call essentially aborts the update process, so none of the security
11 engines will begin to use the new set of rules yet (even those security engines that
12 had returned a value signifying "OK").

13 Fig. 8 is a flowchart illustrating another example process 800 for updating a
14 security policy. Process 800 is implemented by a security engine on a device,
15 such as a security engine 114-118 on host computer 102 of Fig. 1. Process 800
16 may be performed in software, hardware, firmware, or combinations thereof.

17 Initially, a new set of rules and/or data are received for the new policy to be
18 enforced (block 802). As discussed above, the rules and data may be received at
19 substantially the same time, or alternatively the security engine may obtain data
20 from a data store (e.g., the dynamic rules data store discussed herein) as needed.
21 The new rules and/or data are then processed (block 804). Processing of the new
22 rules and/or data creates an internal set of rules for the security engine to follow
23 (e.g., in an internal format of the security engine) in enforcing the new security
24 policy.
25

1 Process 800 proceeds based on whether the processing of the rules was
2 successful (block 806). If the security engine has finished processing the set of
3 rules and/or data that it received and is ready to begin using the new set of rules
4 and/or data (e.g., all that remains is for the security engine to change its pointer to
5 point to the new set of rules rather than the previous set of rules), then the
6 processing was successful. Otherwise, the processing was not successful. If the
7 processing was successful then a value signifying "OK" is returned (block 808).
8 However, if the processing was not successful then a value signifying "Fail" is
9 returned (block 810). The return values in blocks 808 and 810 are returned to a
10 component(s) that is coordinating the updating of the security policy on the host
11 computer (e.g., the rule manager discussed herein).

12 Regardless of the value returned, the security engine continues to use the
13 previous or old set of rules until instructed to rollback or begin using new rules
14 (block 812). If instructed to begin using the new rules, then the security engine
15 begins using the new rules and/or data (block 814), such as by changing a pointer
16 from its previous set of rules to its new set of rules. The instruction to begin using
17 the new rules can be received by the security engine in any of a variety of
18 manners, as discussed above.

19 However, if instructed to rollback, then the security engine discards any
20 results of processing the new rules and/or data (block 816). This discarding can be
21 performed regardless of whether the security engine has finished processing the
22 new set of rules it received.

23 Thus, as can be seen in Fig. 8, the security engine continues to use its
24 previous set of rules until an indication to switch to the new rules is received. By
25 the time such an indication is received, the security engine is ready to begin using

1 the new rules, and very little time is required for the switch to occur. For example,
2 the security engine may simply need to switch a pointer to point to its new set of
3 rules rather than its previous set of rules.

4 5 Application Program Interface (API)

6 An API, such as API 124 discussed above with respect to Fig. 1, permits the
7 communication of security policies and event information among various
8 components and programs (e.g., security engines) in the host computer. In one
9 embodiment the API is defined using the Component Object Model (COM). The
10 API supports methods for loading and unloading security engines, sending
11 security policies to security engines, communicating changes in security policy
12 data to interested security engines, allowing the host user to interact with the
13 security engine at decision-making time to allow or disallow certain policy-
14 specified behaviors, and centralized configuration management for the security
15 engines.

16 The systems and procedures discussed herein enable the security of a
17 computing system to be centrally managed by targeting security policies to
18 a particular computing system or a group of computing systems.
19 Additionally, these systems and procedures collect and correlate events and
20 other information, such as security-related events, generated by those
21 computing systems or other data sources.

22 In one embodiment, the interface supports client access to security
23 policies and event databases via secure, authenticated protocols. The
24 interface permits the communication between various components or
25 application programs and one or more security engines. The interface also

1 defines how the security engines communicate with each other and with
2 other devices, components, services, or programs.

3 In one embodiment, the interface is defined as a COM interface,
4 using a custom loader to reduce the likelihood of an attacker switching
5 security engines for the attacker's own code by changing the COM registry
6 values.

7 In an example embodiment, the function calls supported by the API
8 are:

9 Agent-to-Security Engine Function Calls

- 10 • Initialize
- 11 • Shutdown
- 12 • PreparePolicy
- 13 • CommitPolicy
- 14 • RollbackPolicy
- 15 • WriteData
- 16 • WriteConfig

17 These seven function calls are referred to as the "ISecurityEngine
18 interface".

19
20 Security Engine-to-Agent Function Calls

- 21 • ReadAndRegisterNotifyData
- 22 • WriteSEData
- 23 • UnRegisterNotifyData
- 24 • GetDataAttribute
- 25 • ReadAndRegisterNotifyConfig

1 UnRegisterNotifyConfig

2 QueryUser

3 Complete

4 The first seven function calls above are referred to as the “ISecurityAgent
5 interface” and the last function call (Complete) is referred to as the
6 “IAgentCallback interface”.

7 A function call may also be referred to as a “call”, “method”, “function”, or
8 “service”. Details regarding these function calls are provided below. Alternate
9 embodiments may use additional function calls and/or may omit one or more of
10 the function calls discussed herein.

11 In one embodiment, an agent, such as an event manager or a security
12 agent communicates with the security engines via the API. An agent may
13 also be referred to as a “manager”. In particular embodiments, the agent
14 will not call a particular security engine when an API call is already
15 outstanding. There are exceptions to this rule for the asynchronous API
16 calls. In these cases, the permitted agent actions are defined below in state
17 tables.

18
19 Initialize Function Call

20 This method is called once for each security engine that is known to
21 the agent. The method is called during agent startup. The Initialize
22 function call loads the security engine and allows it to perform initialization
23 operations.

1 This method is called asynchronously in turn for each security
2 engine by the agent, and the callbacks are processed as they are received.

3 The agent will wait for all the callbacks to complete before continuing.

4 This method is defined as follows:

5 HRESULT Initialize(
6 [in] ISecurityAgent *pAgent,
7 [in] IAgentCallback *pCallback);

8 pAgent is a COM interface that can be used by the security engines
9 to call back into the agent.

10 pCallback is the callback object defined below.

11 If the Initialize function call fails, or if it does not return in a
12 reasonable amount of time, then Shutdown will be called. Due to possible
13 race conditions, security engines handle Shutdown before Initialize has
14 returned.
15

16 Shutdown Function Call

17 This method is called once for each security engine that was called
18 to Initialize by the agent. The method allows the security engine to begin
19 its shutdown processing. Even if Initialize failed, the agent calls Shutdown
20 to allow the security engine to close any resources that were allocated. For
21 example, this method allows the security engine to perform a complex
22 shutdown that cannot be performed during DLL_PROCESS_DETACH
23 handling.
24
25

1 Since this method may take a significant amount of time to
2 complete, it uses a callback object to indicate that it has completed
3 shutdown processing. When this method is called as a result of in-process
4 system shutdown, the time available for processing to complete is limited,
5 and the agent may be terminated by the system before the callback is
6 completed.

7 This method is defined as follows:

```
8  
9 typedef enum tagSHUTDOWN_TYPE  
10 {  
11     SHUTDOWN_NORMAL = 0,  
12     SHUTDOWN_SYSTEM  
13 } SHUTDOWN_TYPE;
```

```
14  
15 HRESULT Shutdown(  
16 [in] SHUTDOWN_TYPE eType,  
17 [in] IAgentCallback *pCallback );
```

18 eType is an enumeration of either SHUTDOWN_NORMAL or
19 SHUTDOWN_SYSTEM.

20 pCallback is the callback object defined below.

21 DLL Unload will occur after the Shutdown callback has been made
22 (or the timeout has occurred). Since the callback can be made
23 asynchronously, the agent may have to wait for the thread that made the
24 callback to exit before continuing to unload the security engine DLL. This
25 allows the callback stack frame to unwind to a point outside the DLL that
will be unloaded and avoid an exception in the process. If the callback is

1 made during the Shutdown call, this extra step is not necessary, because the
2 security engine threads are assumed to be shutdown.

3 Errors are logged as operational events but otherwise ignored,
4 because the Agent is about to close down anyway.

6 PreparePolicy Function Call

7 This method is called by the agent when it receives an updated
8 policy. The resulting policies are merged and each RuleSet is built to pass
9 to the correct security engine. The XML data is passed as an IStream
10 object that can be used by MSXML (Microsoft XML) - either DOM
11 (Document Object Model) or SAX (Simple API for XML) - to read the
12 XML data. This method is defined as follows:

13
14 HRESULT PreparePolicy(
15 [in] IStream *pstreamRuleset,
16 [in] IAgentCallback *pCallback);

17 pstreamRuleset is a COM interface to a Stream object that allows
18 reading of the XML Rule Set. This IStream can be assumed to be local to
19 the machine and not accessing data across the network.

20 pCallback is the callback object defined below.

21 If the call returns an error, the security engine is assumed to be
22 continuing to run the previously-applied policy (which may be a boot-time
23 policy). The agent calls a RollbackPolicy for all security engines whose
24 PreparePolicy succeeds, but not for any failing security engine. This
25 process is started as soon as any security engine returns an error. In

1 addition, if the PreparePolicy callback does not arrive in a timely manner,
2 the agent treats this as a failure on the part of that security engine.
3 Therefore, security engines assume that the agent can call RollbackPolicy
4 before the PreparePolicy call has returned.

6 CommitPolicy Function Call

7 This method is called by the agent when all the security engines
8 have returned success on PreparePolicy calls. This call causes them to
9 switch to the new policy. This method is defined as follows:

10
11 HRESULT CommitPolicy(void);

12
13 This call is synchronous, and the agent will wait for one call to
14 complete before moving on to the next call. In one embodiment, it is
15 expected that all the work that could fail a policy update is performed in the
16 PreparePolicy call and this call is a simple switch from the old to the new
17 policy data structures.

18 The CommitPolicy method returns catastrophic failures, such as a
19 failure of communications between User and Kernel parts of the security
20 engine. When this call does return an error, the agent attempts to reload the
21 previous policy and re-apply that policy. Since there was an error, this may
22 not work and the policy enforcement will be left in an unknown state. An
23 operational error will be logged by the agent if CommitPolicy fails.

RollbackPolicy Function Call

This method is called by the agent when a security engine returns an error on its PreparePolicy call. This call causes the agent to call all the other security engines to abort the update and revert to the in-force policy.

This method is defined as follows:

```
HRESULT RollbackPolicy( void );
```

This call is asynchronous because the rollback processing expected of the security engines is extensive, roughly mirroring that in handling PreparePolicy. When the security engine is done processing this call, the security engine calls Complete to indicate the status of the rollback.

If policy is rolled back, any data registered following PreparePolicy is deregistered by the agent – the system is rolled back to the previous set of data subscribed by each security engine. For this reason, the security engines do not discard their local copies of orphaned data until they receive the CommitPolicy call from the agent. The agent is responsible for handling the timing window where ReadAndRegisterData calls from security engines arrive during policy rollback.

RollbackPolicy and the Complete callback may return catastrophic failures. It is expected that security engines implement PreparePolicy such that rollback can be supported. An operational error is logged by the agent when RollbackPolicy fails. No assumption can be made about the state of policy enforcement by that security engine and any cooperating security

1 engines after this happens. Future policy updates will continue to be sent to
2 the security engine.

4 WriteData Function Call

5 This method is called by the agent when a piece of data that the
6 security engine has previously called ReadAndRegisterNotifyData for
7 changes. The parameters are similar to the ReadAndRegisterNotifyData
8 call, except that the memory ownership belongs with the agent, so the
9 security engine must not delete the item once it is processed.

10 WriteData is not called when the security engine is in the process of
11 receiving a new policy from the agent; i.e., between calls to PreparePolicy
12 and CommitPolicy/RollbackPolicy. Any data changes detected by the
13 agent at this time are batched up and sent down to the interested security
14 engines once the new policy has been committed or rolled back. The queue
15 of pending updates is optimized by the agent to avoid as far as possible
16 communicating multiple consecutive changes to the same piece of data.

17 The WriteData method is defined as follows:

```
18  
19 #define DF_DYNAMIC 0x1  
20 #define DF_COLLECTION 0x2  
21 #define DF_BOOLEAN 0x4  
22 #define DF_PERSISTED 0x8  
23  
24  
25
```

```
1      HRESULT WriteData(  
2          [in] REFGUID guidDataID,  
3          [in] DWORD dwFlags,  
4          [in] DWORD dwDataSize,  
5          [in] VARIANT varData,  
6          [in] DWORD dwKeySize,  
7          [in] byte *pbKeyValue);
```

8 The parameters for passing a Key (dwKeySize, pbKeyValue) are
9 used when passing context associated with a previous query back to the
10 security engine. The security engine uses this context to correlate the result
11 of the query with a previous QueryUser call that it issued to the agent. This
12 extra data is necessary because a given query may occur multiply for
13 different contexts in the same rule - for example, asking the user whether
14 application X is allowed to modify a registry value, then later asking the
15 same question about application Y.

16 Errors are logged as Operational Events but otherwise ignored.
17 Future updates to the same piece of data will still be notified to the failing
18 security engine. If the security engine wishes to prevent this, it can call
19 UnRegisterNotifyData for that piece of data.

20 WriteConfig Function Call

21 This method allows the agent to distribute configuration data to
22 interested security engines. Once a security engine has read the
23 configuration data using the ReadAndRegisterNotifyConfig method, it will
24 be informed of changes to that data by the agent calling this method. The
25 method is defined as follows:

1 HRESULT WriteConfig(
2 [in] WCHAR *wszDataName,
3 [in] VARIANT varData);

4 wszDataName is the Text name of the configuration data item being
5 written, and is the name used in the registry for this data.

6 varData is the variant structure that contains the single data item that
7 the name represents. This data can be of various types, depending on what
8 the type of the data is in the registry. The agent does no type checking –
9 the security engine is expected to check the data type as it requires,
10 according to the context.

11 Errors are logged as Operational Events but otherwise ignored.
12 Future updates to the same configuration parameter data will still be
13 notified to the failing security engine. If the security engine wishes to
14 prevent this, it should call UnRegisterNotifyConfig for that piece of data.

15 A particular security engine does not typically call the agent while
16 an API call from that agent is already outstanding. The agent treats this as
17 an error and ignores the second call.

18 19 ReadAndRegisterNotifyData Function Call

20 This method allows a security engine to read data from the dynamic
21 rules data subsystem for use in their rules processing. Once a security
22 engine has read the data, it will be informed of changes to that data by the
23 agent calling the WriteData method of the ISecurityEngine interface. The
24 method is defined as follows:
25

1 HRESULT ReadAndRegisterNotifyData(
2 [in] REFGUID guidDataID,
3 [out] DWORD *pdwFlags,
4 [out] DWORD *pdwDataSize,
5 [out] VARIANT *pvarData);

6 guidDataID is the GUID of the data item to retrieve.

7 pdwFlags is a set of flags that describe the data item. Example
8 values can be DYNAMIC or STATIC as well as COLLECTION or
9 BOOLEAN.

10 pdwDataSize is the total size of the Data Items in the array that is
11 returned in the variant data

12 pvarData is the variant structure that contains a reference to the array
13 of data items, or the data item value for Boolean data types. The variant is
14 empty on input.

15 It is an error for a security engine to ask for data that is no longer in
16 the policy. The agent will generate an Operational Event on any error. In
17 this case, there is no guarantee that the security engine and the agent have a
18 consistent view of the affected data.

19 WriteSEData Function Call

20 This method is called by the security engine when a piece of data
21 changes that the security engine owns and publishes (for persistence, or use
22 by other security engines. The parameters are similar to the WriteData call,
23 except that the memory ownership belongs with the security engine, so the
24 agent does not delete the item once it is processed. The method is defined
25 as follows:

1
2 HRESULT WriteSEData(
3 [in] REFGUID guidDataID,
4 [in] DWORD dwDataSize,
5 [in] VARIANT varData);

6 This method can be called at any time, including while another
7 WriteSEData call is still outstanding, on any thread. It is the agent's
8 responsibility to ensure serialization, if necessary.

9 The owner of a data item is identified in the collection definition by
10 a GUID that defines the owner. This could be the GUID of a security
11 engine or an identifier for the agent, or possibly an identifier for another
12 consumer.

13 If a security engine defines a collection that it owns, it is assumed
14 that the data will be published to the agent via this API.

15 The agent will log any error as an Operational Event. The security
16 engine can decide whether or not to continue providing updates after an
17 error. There is no guarantee that the agent's version of the data is
18 consistent with the security engine's view after an error.

19 UnRegisterNotifyData Function Call

20 This method allows a security engine to stop receiving WriteData
21 notifications for data items it is no longer interested in. The method is
22 defined as follows:
23
24
25

1 HRESULT UnRegisterNotifyData(
2 [in] REFGUID guidDataID);

3 guidDataID is the GUID identifying the data item for which the
4 security engine is no longer interested in change notifications. The security
5 engine can indicate that it wishes to deregister all current notifications by
6 passing in the Null GUID {00000000-0000-0000-000000000000}.

7 The agent will log any error as an Operational Event. This includes
8 the case where the data is not known to the agent, to assist with diagnosis of
9 policy management problems.

10 GetDataAttribute Function Call

11 This method allows a security engine to retrieve a particular attribute
12 associated with a data item. The attribute name is the same as the name
13 that is in the Policy XML, including the case of the text. Attribute values
14 can only change when a policy is changed, so there is no notification
15 system needed for this data. The method is defined as follows:

16
17 HRESULT GetDataAttribute(
18 [in] REFGUID guidDataID,
19 [in] WCHAR *wszAttributeName,
20 [out] VARIANT *pvarAttributeValue);

21 This method can be called at any time.

22 guidDataID is the GUID identifying the data item to retrieve the
23 attribute for.

24 wszAttributeName is the name of the attribute, exactly as it is in the
25 policy document.

1 pvarAttributeValue is the attribute value as a Variant. Normal
2 output parameter allocation rules apply. The agent allocates a new Variant
3 with the information and it is the caller's responsibility to free it later.

5 ReadAndRegisterNotifyConfig Function Call

6 This method allows a security engine to read configuration data from
7 the agent. Once a security engine has read the configuration data, it will be
8 informed of changes to that data by the agent calling the WriteConfig
9 method of the ISecurityEngine interface.

10 Configuration data for the agent and its hosted security engines may
11 be located under a common root. The method is defined as follows:

```
12 HRESULT ReadAndRegisterNotifyConfig(  
13     [in] WCHAR *wszDataName,  
14     [out] VARIANT *pvarData);
```

15 wszDataName is the Text name of the configuration data item to
16 retrieve, and is the name used in the registry for this data. This identifies
17 the individual item relative to the common agent root. No leading '\'
18 character is required. The value is case-insensitive, but whitespace
19 characters are significant.

20 pvarData is the variant structure that contains the single data item
21 that the name represents. This data can be of various types, depending on
22 what the type of the data is in the registry. The agent does no type
23 checking – the security engine is expected to check the data type as it
24 requires, according to the context.

1 The agent will log any error as an Operational Event.

2
3 UnRegisterNotifyConfig Function Call

4 This method allows a security engine to stop receiving WriteConfig
5 notifications for data items it is no longer interested in. The method is
6 defined as follows:

7
8 HRESULT UnRegisterNotifyConfig(
9 [in] WCHAR *wszDataName);

10 wszDataName is the Text name identifying the configuration data
11 item for which the security engine is no longer interested in change
12 notifications.

13 The agent will log any error as an Operational Event. This includes
14 the case where the data is not known to the agent, to assist with diagnosis of
15 configuration management problems.

16
17 QueryUser Function Call

18 This method allows a security engine to ask the agent to display a
19 specific message to the user, returning the answer that the user selected.
20 The agent can also cache this answer, and persist that value over agent re-
21 starts. The question that the user is presented with can contain specific
22 information about why the user is being asked this question. This
23 information can be provided by the security engine and can be different
24 each time this method is called. How the agent decides whether this
25

1 question has been asked before and what the answer is, is determined by the
2 Key Information that the security engine provides.

3 The call returns to the security engine immediately. The security
4 engine then suspends the operation of the session/thread that triggered this
5 query until it is notified of a response. This happens when the user keys in
6 a response, or when the Query times out. The timeout processing is
7 handled by the agent. At this point, the agent updates the relevant data-
8 item with the keyed or default response, and notifies the security engine of
9 the result with its associated context.

10 Since obtaining a response to such queries is time-critical, this API
11 can be called at any time by a security engine that is enforcing a rule
12 requiring a query to be issued. The method is defined as follows:

```
13 HRESULT QueryUser(  
14     [in] REFGUID guidQueryItem,  
15     [in] DWORD dwKeySize,  
16     [in] byte *pbKeyValue,  
17     [in] SAFEARRAY(VARIANT) pvarQueryParams);
```

17 guidQueryItem is the GUID of the data item that contains the base
18 strings that are used to ask the user the question, and provide the possible
19 answers to that question.

20 dwKeySize is the length of the Key Value, in bytes.

21 pbKeyValue is the set of bytes that define the unique key for this
22 query.

23 pvarQueryParams is a Safearray of Variants containing the query
24 parameters to be substituted into the query text that is displayed to the user.
25

1 The order and syntax of the parameters is defined by the rule type with
2 which this QueryUser action is associated.

3 The agent will return an error if the data item is not identifiable.
4 Errors in executing the query will be logged as Operational Events. In this
5 case, the default action is returned to the security engine.
6

7 Complete Function Call

8 This method notifies the agent that a security engine has completed
9 processing associated with a prior asynchronous call from the agent to that
10 security engine. Although a particular security engine can potentially have
11 more than one asynchronous calls from the agent outstanding, the agent
12 manages internal state for each security engine such that the context of a
13 particular Complete callback is unambiguous. The method is defined as
14 follows:
15

```
16 HRESULT Complete(  
17     [in] HRESULT hrCompletionCode);
```

18 hrCompletionCode is the return code for the asynchronous call the agent
19 previously made to this security engine.
20

21 Interface Usage

22 The following describes example restrictions on how these APIs are
23 used to interact with one or more security engines.
24
25

At a particular time, the security engine is in a certain state with respect to its interactions with the agent. The following list identifies possible security engine states.

State	Definition
Pending_Initialize	The security engine DLL has been loaded but no API calls received yet. The Policy state at this point depends on the security engine – NSE has a boot-time policy, behavioral blocking has none until it is given rules by the agent.
Initializing	Initialize has been called but not completed
Running	The security engine has called back the agent to say it Initialized successfully, and is enforcing either (initially) boot-time or (after subsequent CommitPolicy) agent-supplied policy
Preparing_Policy	PreparePolicy has been called but no callback has happened
Policy_Prepared	PreparePolicy callback completed with success return code, waiting for CommitPolicy call
Policy_Rollback	Security engine called with RollbackPolicy, processing the rollback request
Shutting_Down	Shutdown has been called but not completed
Pending_Termination	Shutdown complete - waiting for process termination

1
2 The permitted interactions between the agent and security engines
3 can be formalized as a set of tables which define the APIs that can be called
4 by each entity when a security engine is in a particular state, and what state
5 change or other action needs to be taken by the security engine as a result.

6 The operational state of the agent is assumed to be unimportant – the
7 security engines can assume it remains in normal operation at all times
8 while the security engines are loaded into memory.

9 The state tables cover the following phases of the security engine's
10 lifecycle:

- 11 • Initialization
- 12 • Policy Update from agent
- 13 • Shutdown

14
15 Any combination of API call and security engine state not covered in
16 these tables can be considered a misuse of the API. It is the responsibility
17 of the API caller to avoid such misuse.

18 The following state table defines the permitted sequences of APIs
19 during security engine initialization, and security engine state changes
20 according to inputs from the agent. A call to any API not listed as a
21 permitted input for the list of states associated with security engine
22 initialization implies a protocol error on the calling entity's part.

Security Engine State Agent API Calls	Pending_Initialize	Initializing
Initialize	Initializing	ERROR
Shutdown	ERROR	Pending_Termination
WriteConfig	ERROR	OK
Security Engine API Calls		
Complete(OK)	ERROR	Running (no policy)
Complete(FAIL)	ERROR	Pending_Termination
ReadAndRegisterNotify Config	ERROR	OK

The following state table defines the permitted sequences of APIs during policy update, and the associated security engine state changes. A call to any API not listed as a permitted input here for the list of states associated with policy update implies a protocol error on the calling entity's part.

Security Engine State Agent API Calls	Running	Preparing_Policy
PreparePolicy	Preparing_Policy	ERROR
WriteConfig	OK	OK
Security Engine API Calls		
Complete(OK)	ERROR	Policy_Prepared
Complete(FAIL)	ERROR	Running (old policy)
ReadAndRegisterNotifyConfig	OK	OK
ReadAndRegisterNotifyData	OK	OK

Security Engine State Agent API Calls	Policy_Prepared
CommitPolicy	Running (new policy)
RollbackPolicy	Policy_Rollback (old policy)
WriteConfig	OK
Security Engine API	

Calls	
Complete(OK)	ERROR
Complete(FAIL)	ERROR
ReadAndRegisterConfig	OK
ReadAndRegisterData	OK

An example of the overall policy update sequence, taking into account multiple hosted security engines, is described below.

1. Each security engine's PreparePolicy is called.
2. The agent waits for each security engine to call Complete with Success or a Failure
3. If any security engine reports a failure, every other security engine will have their RollbackPolicy method called.
4. If no security engine reports a failure, the CommitPolicy method is called for each security engine.
5. If another failure is discovered, or if a Shutdown is necessary, before any CommitPolicy methods are called, the RollbackPolicy method is called for each security engine.

The following state table defines the permitted sequences of APIs during security engine shutdown, and security engine state changes according to inputs from the agent. A call to any API not listed as a permitted input here for the list of states associated with security engine shutdown implies a protocol error on the calling entity's part.

SE State Agent API Calls	Initializing, Running, Preparing_Policy, Policy_Prepared, Policy_Rollback	Shutting_Down
Shutdown	Shutting_Down	ERROR
SE API Calls		
Complete	ERROR	Pending_Termination

Listed below are example collection types supported by the agent, and descriptions regarding how each collection is passed as dynamic data through the ReadAndRegisterNotifyData and WriteData method calls.

Many of the data items discussed below can be handled by passing a single BSTR string, or packing unsigned integers into a LONG or a LONGLONG variant type. Those items that don't easily fit this model are: DirectorySet, ProtocolSet and IPv4AddressSet. For each of these types a packing system is suggested that packs the data into a BSTR string to allow easy transfer in a SafeArray.

FileSet

Data passed for each item:

Filename – string

Implementation:

BSTR

DirectorySet

Data passed for each item:

Directory Name – String

Recursive – Flag

Implementation:

Packed BSTR - “Recursive Flag:String”

Recursive Flag is a single character –

‘R’ – Recursive

‘F’ – Flat

RegistrySet

Data passed for each item:

Registry Key Name – String

Implementation:

Packed BSTR - “Recursive Flag:String”

Recursive Flag is a single character –

‘R’ – Recursive

‘F’ – Flat

1 Protocol

2 Data passed for each item:

3 Primary / Secondary – String or Enumeration

4 IP Type – String or Enumeration

5 Direction - String or Enumeration

6 Port or Port Range – One or Two Integers (16 bit, unsigned
7 integers)

8
9 Implementation:

10 Packed LONGLONG:

11 1 Byte - Primary / Secondary

12 1 Byte – IP Type TCP/UDP

13 1 Byte – Direction In/Out/Both

14 1 Byte – Unused

15 2 Bytes – Port Range End (or Zero)

16 2 Bytes – Port Range Start (or Port)

17
18 ProcessSet

19 Data passed for each item:

20 Process Name or Path – String

21
22 Implementation:

23 BSTR

1
2 NetworkPortSet

3 Data passed for each item:

4 Port or Port Range – One or Two Integers (16 bit, unsigned
5 integers)

6
7 Implementation:

8 Packed LONG: Start = Low Word, End = High Word.

9 High Word is Zero if not a Port Range

10
11 NetworkIPv4AddressSet

12 Data passed for each item:

13 One of:

14 IPv4 Address – String (can contain wildcards)

15 IPv4 Address Range – 2 Strings

16 FQDN – String

17 Hostname – String

18
19 Implementation:

20 Packed BSTR: “T:String 1:String 2”

21 T – Type - One Character for Address, Address Range,
22 HostName or FQDN

23 String 1 – Address, Start Address, HostName or
24 FQDN

25 String 2 – End Address for Address Range

1
2 UserSet

3 Data passed for each item:

4 User Account Name – String

5
6 Implementation:

7 BSTR

8
9 UserGroupSet

10 Data passed for each item:

11 User Group Name – String

12
13 Implementation:

14 BSTR

15
16 FileOpSet

17 Data passed for each item:

18 File Operation – String (or Enumeration)

19
20 Implementation:

21 BSTR

1 DirOpSet

2 Data passed for each item:

3 Directory Operation – String (or Enumeration)

4
5 Implementation:

6 BSTR

7
8 ProcessOpSet

9 Data passed for each item:

10 Process Operation – String (or Enumeration)

11
12 Implementation:

13 BSTR

14
15 RegKeyOpSet

16 Data passed for each item:

17 Registry Key Operation – String (or Enumeration)

18
19 Implementation:

20 BSTR

1 RegValueOpSet

2 Data passed for each item:

3 Registry Value Operation – String (or Enumeration)

4
5 Implementation:

6 BSTR

7
8 UserOpSet

9 Data passed for each item:

10 User Account Operation – String (or Enumeration)

11
12 Implementation:

13 BSTR

14
15 UserGroupOpSet

16 Data passed for each item:

17 User Group Operation – String (or Enumeration)

18
19 Implementation:

20 BSTR

1 JobOpSet

2 Data passed for each item:

3 Job Operation – String (or Enumeration)

4
5 Implementation:

6 BSTR

7
8 Generic

9 Data passed for each item:

10 Value – String

11
12 Implementation:

13 BSTR

14
15 QuerySet

16 To the security engine, QuerySet appears as a single item collection
17 that contains the result of a query to the user. The associated context is
18 passed to the security engine as a separate parameter. The internal structure
19 of the QuerySet is not typically needed by a security engine, only the
20 context and the query result.
21
22
23
24
25

Boolean (boolDefine)

Data passed for the single item:

Boolean – True or False

Implementation:

LONG - False = 0, True = 1

Fig. 9 illustrates a general computer environment 900, which can be used to implement the techniques described herein. The computer environment 900 is only one example of a computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the computer and network architectures. Neither should the computer environment 900 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the example computer environment 900.

Computer environment 900 includes a general-purpose computing device in the form of a computer 902. One or more media player applications can be executed by computer 902. The components of computer 902 can include, but are not limited to, one or more processors or processing units 904 (optionally including a cryptographic processor or co-processor), a system memory 906, and a system bus 908 that couples various system components including the processor 904 to the system memory 906.

The system bus 908 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a point-to-point connection, a switching fabric, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. By way of

1 example, such architectures can include an Industry Standard Architecture (ISA)
2 bus, a Micro Channel Architecture (MCA) bus, an Enhanced ISA (EISA) bus, a
3 Video Electronics Standards Association (VESA) local bus, and a Peripheral
4 Component Interconnects (PCI) bus also known as a Mezzanine bus.

5 Computer 902 typically includes a variety of computer readable media.
6 Such media can be any available media that is accessible by computer 902 and
7 includes both volatile and non-volatile media, removable and non-removable
8 media.

9 The system memory 906 includes computer readable media in the form of
10 volatile memory, such as random access memory (RAM) 910, and/or non-volatile
11 memory, such as read only memory (ROM) 912. A basic input/output system
12 (BIOS) 914, containing the basic routines that help to transfer information
13 between elements within computer 902, such as during start-up, is stored in ROM
14 912. RAM 910 typically contains data and/or program modules that are
15 immediately accessible to and/or presently operated on by the processing unit 904.

16 Computer 902 may also include other removable/non-removable,
17 volatile/non-volatile computer storage media. By way of example, Fig. 9
18 illustrates a hard disk drive 916 for reading from and writing to a non-removable,
19 non-volatile magnetic media (not shown), a magnetic disk drive 918 for reading
20 from and writing to a removable, non-volatile magnetic disk 920 (e.g., a “floppy
21 disk”), and an optical disk drive 922 for reading from and/or writing to a
22 removable, non-volatile optical disk 924 such as a CD-ROM, DVD-ROM, or other
23 optical media. The hard disk drive 916, magnetic disk drive 918, and optical disk
24 drive 922 are each connected to the system bus 908 by one or more data media
25 interfaces 925. Alternatively, the hard disk drive 916, magnetic disk drive 918,

1 and optical disk drive 922 can be connected to the system bus 908 by one or more
2 interfaces (not shown).

3 The disk drives and their associated computer-readable media provide non-
4 volatile storage of computer readable instructions, data structures, program
5 modules, and other data for computer 902. Although the example illustrates a hard
6 disk 916, a removable magnetic disk 920, and a removable optical disk 924, it is to
7 be appreciated that other types of computer readable media which can store data
8 that is accessible by a computer, such as magnetic cassettes or other magnetic
9 storage devices, flash memory cards, CD-ROM, digital versatile disks (DVD) or
10 other optical storage, random access memories (RAM), read only memories
11 (ROM), electrically erasable programmable read-only memory (EEPROM), and
12 the like, can also be utilized to implement the example computing system and
13 environment.

14 Any number of program modules can be stored on the hard disk 916,
15 magnetic disk 920, optical disk 924, ROM 912, and/or RAM 910, including by
16 way of example, an operating system 926, one or more application programs 928,
17 other program modules 930, and program data 932. Each of such operating
18 system 926, one or more application programs 928, other program modules 930,
19 and program data 932 (or some combination thereof) may implement all or part of
20 the resident components that support the distributed file system.

21 A user can enter commands and information into computer 902 via input
22 devices such as a keyboard 934 and a pointing device 936 (e.g., a "mouse").
23 Other input devices 938 (not shown specifically) may include a microphone,
24 joystick, game pad, satellite dish, serial port, scanner, and/or the like. These and
25 other input devices are connected to the processing unit 904 via input/output

1 interfaces 940 that are coupled to the system bus 908, but may be connected by
2 other interface and bus structures, such as a parallel port, game port, or a universal
3 serial bus (USB).

4 A monitor 942 or other type of display device can also be connected to the
5 system bus 908 via an interface, such as a video adapter 944. In addition to the
6 monitor 942, other output peripheral devices can include components such as
7 speakers (not shown) and a printer 946 which can be connected to computer 902
8 via the input/output interfaces 940.

9 Computer 902 can operate in a networked environment using logical
10 connections to one or more remote computers, such as a remote computing device
11 948. By way of example, the remote computing device 948 can be a personal
12 computer, portable computer, a server, a router, a network computer, a peer device
13 or other common network node, game console, and the like. The remote
14 computing device 948 is illustrated as a portable computer that can include many
15 or all of the elements and features described herein relative to computer 902.

16 Logical connections between computer 902 and the remote computer 948
17 are depicted as a local area network (LAN) 950 and a general wide area network
18 (WAN) 952. Such networking environments are commonplace in offices,
19 enterprise-wide computer networks, intranets, and the Internet.

20 When implemented in a LAN networking environment, the computer 902 is
21 connected to a local network 950 via a network interface or adapter 954. When
22 implemented in a WAN networking environment, the computer 902 typically
23 includes a modem 956 or other means for establishing communications over the
24 wide network 952. The modem 956, which can be internal or external to computer
25 902, can be connected to the system bus 908 via the input/output interfaces 940 or

1 other appropriate mechanisms. It is to be appreciated that the illustrated network
2 connections are exemplary and that other means of establishing communication
3 link(s) between the computers 902 and 948 can be employed.

4 In a networked environment, such as that illustrated with computing
5 environment 900, program modules depicted relative to the computer 902, or
6 portions thereof, may be stored in a remote memory storage device. By way of
7 example, remote application programs 958 reside on a memory device of remote
8 computer 948. For purposes of illustration, application programs and other
9 executable program components such as the operating system are illustrated herein
10 as discrete blocks, although it is recognized that such programs and components
11 reside at various times in different storage components of the computing device
12 902, and are executed by the data processor(s) of the computer.

13 Various modules and techniques may be described herein in the general
14 context of computer-executable instructions, such as program modules, executed
15 by one or more computers or other devices. Generally, program modules include
16 routines, programs, objects, components, data structures, etc. that perform
17 particular tasks or implement particular abstract data types. Typically, the
18 functionality of the program modules may be combined or distributed as desired in
19 various embodiments.

20 An implementation of these modules and techniques may be stored on or
21 transmitted across some form of computer readable media. Computer readable
22 media can be any available media that can be accessed by a computer. By way of
23 example, and not limitation, computer readable media may comprise "computer
24 storage media" and "communications media."

1 “Computer storage media” includes volatile and non-volatile, removable
2 and non-removable media implemented in any method or technology for storage
3 of information such as computer readable instructions, data structures, program
4 modules, or other data. Computer storage media includes, but is not limited to,
5 RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM,
6 digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic
7 tape, magnetic disk storage or other magnetic storage devices, or any other
8 medium which can be used to store the desired information and which can be
9 accessed by a computer.

10 “Communication media” typically embodies computer readable
11 instructions, data structures, program modules, or other data in a modulated data
12 signal, such as carrier wave or other transport mechanism. Communication media
13 also includes any information delivery media. The term “modulated data signal”
14 means a signal that has one or more of its characteristics set or changed in such a
15 manner as to encode information in the signal. By way of example, and not
16 limitation, communication media includes wired media such as a wired network or
17 direct-wired connection, and wireless media such as acoustic, RF, infrared, and
18 other wireless media. Combinations of any of the above are also included within
19 the scope of computer readable media.

20 Notionally, a programming interface may be viewed generically, as shown
21 in Fig. 10 or Fig. 11. Fig. 10 illustrates an interface Interface1 as a conduit
22 through which first and second code segments communicate. Fig. 11 illustrates an
23 interface as comprising interface objects I1 and I2 (which may or may not be part
24 of the first and second code segments), which enable first and second code
25 segments of a system to communicate via medium M. In the view of Fig. 11, one

1 may consider interface objects I1 and I2 as separate interfaces of the same system
2 and one may also consider that objects I1 and I2 plus medium M comprise the
3 interface. Although Figs. 10 and 11 show bi-directional flow and interfaces on
4 each side of the flow, certain implementations may only have information flow in
5 one direction (or no information flow as described below) or may only have an
6 interface object on one side. By way of example, and not limitation, terms such as
7 application programming or program interface (API), entry point, method,
8 function, subroutine, remote procedure call, and component object model (COM)
9 interface, are encompassed within the definition of programming interface.

10 Aspects of such a programming interface may include the method whereby
11 the first code segment transmits information (where "information" is used in its
12 broadest sense and includes data, commands, requests, etc.) to the second code
13 segment; the method whereby the second code segment receives the information;
14 and the structure, sequence, syntax, organization, schema, timing and content of
15 the information. In this regard, the underlying transport medium itself may be
16 unimportant to the operation of the interface, whether the medium be wired or
17 wireless, or a combination of both, as long as the information is transported in the
18 manner defined by the interface. In certain situations, information may not be
19 passed in one or both directions in the conventional sense, as the information
20 transfer may be either via another mechanism (e.g., information placed in a buffer,
21 file, etc. separate from information flow between the code segments) or non-
22 existent, as when one code segment simply accesses functionality performed by a
23 second code segment. Any or all of these aspects may be important in a given
24 situation, e.g., depending on whether the code segments are part of a system in a
25

1 loosely coupled or tightly coupled configuration, and so this list should be
2 considered illustrative and non-limiting.

3 This notion of a programming interface is known to those skilled in the art
4 and is clear from the foregoing detailed description of the invention. There are,
5 however, other ways to implement a programming interface, and, unless expressly
6 excluded, these too are intended to be encompassed by the claims set forth at the
7 end of this specification. Such other ways may appear to be more sophisticated or
8 complex than the simplistic view of Figs. 10 and 11, but they nonetheless perform
9 a similar function to accomplish the same overall result. We will now briefly
10 describe some illustrative alternative implementations of a programming interface.

11 12 Factoring

13 A communication from one code segment to another may be accomplished
14 indirectly by breaking the communication into multiple discrete communications.
15 This is depicted schematically in Figs. 12 and 13. As shown, some interfaces can
16 be described in terms of divisible sets of functionality. Thus, the interface
17 functionality of Figs. 10 and 11 may be factored to achieve the same result, just as
18 one may mathematically provide 24, or 2 times 2 times 3 times 2. Accordingly, as
19 illustrated in Fig. 12, the function provided by interface Interface1 may be
20 subdivided to convert the communications of the interface into multiple interfaces
21 Interface1A, Interface 1B, Interface 1C, etc. while achieving the same result. As
22 illustrated in Fig. 13, the function provided by interface I1 may be subdivided into
23 multiple interfaces I1a, I1b, I1c, etc. while achieving the same result. Similarly,
24 interface I2 of the second code segment which receives information from the first
25 code segment may be factored into multiple interfaces I2a, I2b, I2c, etc. When

1 factoring, the number of interfaces included with the 1st code segment need not
2 match the number of interfaces included with the 2nd code segment. In either of
3 the cases of Figs. 12 and 13, the functional spirit of interfaces Interface1 and I1
4 remain the same as with Figs. 10 and 11, respectively. The factoring of interfaces
5 may also follow associative, commutative, and other mathematical properties such
6 that the factoring may be difficult to recognize. For instance, ordering of
7 operations may be unimportant, and consequently, a function carried out by an
8 interface may be carried out well in advance of reaching the interface, by another
9 piece of code or interface, or performed by a separate component of the system.
10 Moreover, one of ordinary skill in the programming arts can appreciate that there
11 are a variety of ways of making different function calls that achieve the same
12 result.

13 14 Redefinition

15 In some cases, it may be possible to ignore, add or redefine certain aspects
16 (e.g., parameters) of a programming interface while still accomplishing the
17 intended result. This is illustrated in Figs. 14 and 15. For example, assume
18 interface Interface1 of Fig. 10 includes a function call *Square(input, precision,*
19 *output)*, a call that includes three parameters, *input, precision and output*, and
20 which is issued from the 1st Code Segment to the 2nd Code Segment. If the middle
21 parameter *precision* is of no concern in a given scenario, as shown in Fig. 14, it
22 could just as well be ignored or even replaced with a *meaningless* (in this
23 situation) parameter. One may also add an *additional* parameter of no concern. In
24 either event, the functionality of square can be achieved, so long as output is
25 returned after input is squared by the second code segment. *Precision* may very

1 well be a meaningful parameter to some downstream or other portion of the
2 computing system; however, once it is recognized that *precision* is not necessary
3 for the narrow purpose of calculating the square, it may be replaced or ignored.
4 For example, instead of passing a valid *precision* value, a meaningless value such
5 as a birth date could be passed without adversely affecting the result. Similarly, as
6 shown in Fig. 15, interface I1 is replaced by interface I1', redefined to ignore or
7 add parameters to the interface. Interface I2 may similarly be redefined as
8 interface I2', redefined to ignore unnecessary parameters, or parameters that may
9 be processed elsewhere. The point here is that in some cases a programming
10 interface may include aspects, such as parameters, that are not needed for some
11 purpose, and so they may be ignored or redefined, or processed elsewhere for
12 other purposes.

13 14 Inline Coding

15 It may also be feasible to merge some or all of the functionality of two
16 separate code modules such that the "interface" between them changes form. For
17 example, the functionality of Figs. 10 and 11 may be converted to the functionality
18 of Figs. 16 and 17, respectively. In Fig. 16, the previous 1st and 2nd Code
19 Segments of Fig. 10 are merged into a module containing both of them. In this
20 case, the code segments may still be communicating with each other but the
21 interface may be adapted to a form which is more suitable to the single module.
22 Thus, for example, formal Call and Return statements may no longer be necessary,
23 but similar processing or response(s) pursuant to interface Interface1 may still be
24 in effect. Similarly, shown in Fig. 17, part (or all) of interface I2 from Fig. 11 may
25 be written inline into interface I1 to form interface I1". As illustrated, interface I2

1 is divided into I2a and I2b, and interface portion I2a has been coded in-line with
2 interface I1 to form interface I1". For a concrete example, consider that the
3 interface I1 from Fig. 11 performs a function call square (*input*, *output*), which is
4 received by interface I2, which after processing the value passed with *input* (to
5 square it) by the second code segment, passes back the squared result with *output*.
6 In such a case, the processing performed by the second code segment (squaring
7 *input*) can be performed by the first code segment without a call to the interface.

8 9 Divorce

10 A communication from one code segment to another may be accomplished
11 indirectly by breaking the communication into multiple discrete communications.
12 This is depicted schematically in Figs. 18 and 19. As shown in Fig. 18, one or
13 more piece(s) of middleware (Divorce Interface(s), since they divorce
14 functionality and / or interface functions from the original interface) are provided
15 to convert the communications on the first interface, Interface1, to conform them
16 to a different interface, in this case interfaces Interface2A, Interface2B and
17 Interface2C. This might be done, e.g., where there is an installed base of
18 applications designed to communicate with, say, an operating system in
19 accordance with an Interface1 protocol, but then the operating system is changed
20 to use a different interface, in this case interfaces Interface2A, Interface2B and
21 Interface2C. The point is that the original interface used by the 2nd Code Segment
22 is changed such that it is no longer compatible with the interface used by the 1st
23 Code Segment, and so an intermediary is used to make the old and new interfaces
24 compatible. Similarly, as shown in Fig. 19, a third code segment can be
25 introduced with divorce interface DI1 to receive the communications from

1 interface I1 and with divorce interface DI2 to transmit the interface functionality
2 to, for example, interfaces I2a and I2b, redesigned to work with DI2, but to
3 provide the same functional result. Similarly, DI1 and DI2 may work together to
4 translate the functionality of interfaces I1 and I2 of Fig. 11 to a new operating
5 system, while providing the same or similar functional result.

6 7 Rewriting

8 Yet another possible variant is to dynamically rewrite the code to replace
9 the interface functionality with something else but which achieves the same
10 overall result. For example, there may be a system in which a code segment
11 presented in an intermediate language (e.g. Microsoft IL, Java ByteCode, etc.) is
12 provided to a Just-in-Time (JIT) compiler or interpreter in an execution
13 environment (such as that provided by the .Net framework, the Java runtime
14 environment, or other similar runtime type environments). The JIT compiler may
15 be written so as to dynamically convert the communications from the 1st Code
16 Segment to the 2nd Code Segment, i.e., to conform them to a different interface as
17 may be required by the 2nd Code Segment (either the original or a different 2nd
18 Code Segment). This is depicted in Figs. 20 and 21. As can be seen in Fig. 20,
19 this approach is similar to the Divorce scenario described above. It might be done,
20 e.g., where an installed base of applications are designed to communicate with an
21 operating system in accordance with an Interface 1 protocol, but then the operating
22 system is changed to use a different interface. The JIT Compiler could be used to
23 conform the communications on the fly from the installed-base applications to the
24 new interface of the operating system. As depicted in Fig. 21, this approach of
25

1 dynamically rewriting the interface(s) may be applied to dynamically factor, or
2 otherwise alter the interface(s) as well.

3 It is also noted that the above-described scenarios for achieving the same or
4 similar result as an interface via alternative embodiments may also be combined in
5 various ways, serially and/or in parallel, or with other intervening code. Thus, the
6 alternative embodiments presented above are not mutually exclusive and may be
7 mixed, matched and combined to produce the same or equivalent scenarios to the
8 generic scenarios presented in Figs. 10 and 11. It is also noted that, as with most
9 programming constructs, there are other similar ways of achieving the same or
10 similar functionality of an interface which may not be described herein, but
11 nonetheless are represented by the spirit and scope of the invention, i.e., it is noted
12 that it is at least partly the functionality represented by, and the advantageous
13 results enabled by, an interface that underlie the value of an interface.

14 Although the description above uses language that is specific to structural
15 features and/or methodological acts, it is to be understood that the invention
16 defined in the appended claims is not limited to the specific features or acts
17 described. Rather, the specific features and acts are disclosed as exemplary forms
18 of implementing the invention.